NAME

Embody − Environment Modules Build system

SYNOPSIS

./EMBODY [options]

DESCRIPTION

Embody (Environment Modules Build) is a software build tool with integrated support for the environment-modules package. The tool eases and automates the task of building and installing software packages from source or binary distributions, as well as the management of associated modulefiles.

Embody provides a framework to script the tasks that are customarily described in README and INSTALL files, run these tasks in order or individually, and capture their output in log files.

The design goal was to reduce routine installation tasks to defining variables and shell functions for the key tasks, thereby providing a self-documenting and unified skeleton for maintaining package installations. While there is some conceptual overlap with *rpm* (8), the goal is simplicity and decoupling from rpm's dependencies and database, which enables coexistence of several builds. Useful on HPC systems, new builds can be deployed centrally to shared file systems and without affecting running jobs.

OPERATION

Embody consists of a library *libembody* and a user-defined package-specific script named *EMBODY* by convention. Both are written in *bash* (1).

Package placement

With **Embody**, software is normally deployed into package-specific directories, typically having subdirectories like *bin*, *lib*, and *man*, as determined by the package's native install procedure. This structure will allow several versions and builds to coexist.

The name of the top-level directory is generated in a variable `$prefix`, which is constructed roughly as:

```
$PACKAGE_ROOT/$NAME-$VERSION-$BUILD
```

where the constituent variables are defined by the user in *EMBODY* and by site-defaults in *libembody*. A *modulefile* (5) is automatically created and placed in

```
$MODULE_ROOT/$NAME/$VERSION-$BUILD
```

If, during modulefile installation, a modulefile from a prior version exists in `$MODULE_ROOT/$NAME/`, a *.version* file is created if it does not already exist, so as to prevent premature use of the new build by user's shells. The site administrator can later edit or remove this *.version* file (see ‘‘Modulefile management’’ options), preferably after users have been notified of the upgrade.

The user running *EMBODY* must have write permission in `$PACKAGE_ROOT` and `$MODULE_ROOT`. With a proper setup, such as one employing group permissions, it is often not necessary to run, and in particular install, as root.

Staging Functions

Package deployment is done by a series of so-called staging functions in *bash* (1) syntax. Default functions are pre−defined, and may be re-defined by the user in the *EMBODY* script. The predefined functions detect a couple of deployment styles and execute the canonical action as described below under OPTIONS. The recognized styles are, in this order:

- *rpmbuild* (8) from a *spec* file
- Python-style setup.py
- GNU-style configure + make

The functions and their correspondence to options are:

Function name	Option	Notes	Provided?
stage_download	--download	(1)	no
stage_zap	-z	(2)	no
stage_extract	-x	(1)	no
stage_remove	-r	(3)	yes
stage_uninstall	-u		yes
stage_distclean	-d		yes
stage_prep	-p		yes
stage_build	-b		yes
stage_install	-i		yes
stage_install_aux	-a		yes
stage_module	-m		yes
stage_test	-t		yes
stage_clean	-c	(4)	yes
embody_stages		(5)	
embody_wipe	-w	(5)	

```
(1) Normally runs only once unless --force is given.
(2) Normally runs only as part of stage_extract.
(3) Normally runs only as part of stage_uninstall.
(4) Not run as part of default sequence.
(5) Not a staging function - do not redfine.
```

Unless any of the specific options above are given to *EMBODY* to explicitly pick one or more stages, all staging functions above except *stage_clean* are run in sequence, as hardcoded in the *embody_stages* sequencing function.

A build-specific directory is created in the package source tree to hold log files and (eventually) a test directory:

```
embody-$VERSION-$BUILD/
```

The output of each individual stage is logged into:

```
embody-$VERSION-$BUILD/<stagename>.log
```

and the output of the whole *EMBODY* run is logged into:

```
embody-$VERSION-$BUILD/last.log
```

These files, as indeed the entire source and build directory where *EMBODY* runs, can be left after the build should a problem arise in production. Calling the **−w** option, however, will remove all builds' log dirs.

The EMBODY script

The user creates the *EMBODY* script to reside in a typically version-specific work directory for a package. The name can be anything, but *EMBODY* sorts before *README* or *INSTALL* and stands out.

The script must do the following:

- set package-related variables (`NAME`, `VERSION`, `BUILD`),
- set variables for modulefile content (`MODULE_WHATIS`, `MODULE_HELP`, etc.),
- load the *embody* module and any modules that are prerequisite for the current package,
- source the *embody* library,
- (re−)define zero or more staging functions, and finally,
- run the *embody_stages* sequencing function, the last and main executable statement.

Most of the script will be ‘‘merely’’ definitions of variables and staging functions.

Variables in the EMBODY script

The following variables are expected to be set in the *EMBODY* script:

* Package definition

`NAME`	Package name, without version and build tags. Acceptable characters are letters (possibly in mixed case), numerals, and dashes ‘‘−’’. Underscore ‘‘_’’ is discouraged, and any other ‘‘funny’’ characters are disallowed.
`VERSION`	Package version [optional]. Should consist of numerals, dot ‘‘.’’, and letters.
`BUILD`	Build tag [optional]; can be arbitrarily long. Acceptable characters as in `NAME`.
`BUILD_MULTI`	A multi-line build specification (see MULTI-BUILDS below). Ignored when `BUILD` is set.
`SPECFILE`	name of an *rpm* (8) specfile. The variables `NAME`, `VERSION`, `BUILD`, `MODULE_WHATIS`, and `MODULE_HELP` are set from contents of the spec file, but may be overridden.

* Site defaults

The following are normally predefined in the site’s libembody file:

`PACKAGE_ROOT`	base directory for packages
`MODULE_ROOT`	base directory for modulefiles, default: `$PACKAGE_ROOT/modulefiles`

* Modulefile help items

These following are converted to `proc ModulesHelp` and `module−whatis`, respectively:

`MODULE_WHATIS`	whatis string (should be one line) − required. If this value is missing, the modulefile creation will be skipped.
`MODULE_HELP`	Help text, may be several lines.

* Modulefile contents

These are placed verbatim into the modulefile (leading spaces are stripped):

`MODULE_DEP`	Zero or more `conflict foo` or `prereq foo`
`MODULE_CORE`	The bulk part of the modulefile, `prepend PATH` *etc.*
`MODULE_AUX`	Package-specific auxiliary definitions.

The staging functions have access to all of these variables.

Automatisms

1. `NAME` and `VERSION` are actually optional and are guessed from the package directory if it is named in the customary form *name−x.y.z*. Directories of the form *name−x[.y[.z]][−more]* are also recognized.
2. If `MODULE_CORE` is left empty, it is *guessed* based on the existence of subdirs found in `$prefix/` after *stage_install*. A complete such guess is equivalent to the following:

```
MODULE_CORE="
    prepend-path    PATH            \$prefix/bin
    prepend-path    MANPATH         \$prefix/man
    prepend-path    MANPATH         \$prefix/share/man
    prepend-path    PYTHON_PATH     \$prefix/lib/python
    prepend-path    PYTHON_PATH     \$prefix/lib64/python
    prepend-path    LD_LIBRARY_PATH \$prefix/lib
    prepend-path    LD_LIBRARY_PATH \$prefix/lib64
    prepend-path    INCLUDE         \$prefix/include
"
```

3. For convenience, an environment variable `<NAME>_HOME` is automatically added:

   ```
   setenv <NAME>_HOME  $prefix
   ```

   This is a customary installation requirement for many packages, and also gives users a uniform namespace to access the active package, e.g. `$FOO_HOME/share/`. `<NAME>` is the uppercased value of `$NAME`, with − replaced by _.

OPTIONS

Stage selection

The following options select one or more *staging functions*. Without an explicit selection, most staging functions are executed in the order shown in the table above, subject to the conditions noted. The output fo each stage function is logged under `embody_logdir`/name.log.

−−download Download source files into a local cache. Has effect only if the user defined a `stage_download` function (no default). There is no short option because I ran out of convenient letters.

Recommendations:

* Put downloads into a directory above the version-specific current working directory, such as ../dist . This will avoid re-downloads and simplifies cleanup operations.
* Define variables in the preamable of *EMBODY* to refer to the downloaded files in stage_download and stage_extract.

−z, −−zap Remove source files, i.e., clean the working directory. Has effect only if the user defined a `stage_zap` function (no default).

−x, −−extract Unpack source files; implies **−−zap**. Has effect only if the user defined a `stage_extract` function (no default).

−u, −−uninstall Uninstall the package and remove its modulefile; implies **−−remove** (see below).

−d, −−distclean Perform distclean stage; default: `make distclean` or `setup.py clean`.

−p, −−prep Perform prep stage; default: `./configure`, NOP for `setup.py`.

−b, −−build Perform build stage; default: `make` or `setup.py build`.

−i, −−install Install; default: `make install` or `setup.py install`

−a, −−aux Install auxiliary files; no default.

Experimental: Prior to the actual call to *stage_install_aux*, the current *EMBODY* script will be preserved in `$prefix/` as `.EMBODY`, and the build directory will be symlinked as `.src`.

−m, −−module Install the modulefile.

−t, −−test Perform a test; default: `make check` or `make test` (depending on Makefile); `test.py` for python. Prior to running *stage_test*, the new modulefile will be loaded.

−c, −−clean	Perform cleanup; default: `make clean` or `setup.py clean`.
−X, −U, −D, −P, −B, −I, −A, −M, −T	Perform the stages in the usual order *up to* the given stage. In fact, **−T** is equivalent to the default sequence.

Modulefile management

−e, −−edit	Edit the modulefile.
−l, −−list	List installed module versions and show the contents of *.version*, if it exists. Option **−v** gives more details.
−r, −−remove	Remove the *.version* file, thereby making the lexicographically latest modulefile the default module. (Note that this can produce incorrect behavior when a version number component changes from .9 to .10 .) With **−−force**, also remove the modulefile corresponding to the current `NAME/VERSION−BUILD` triple.
−s, −−show	Construct and show the modulefile, but do not install.

Control

−1, −2, −3, ...	(any numeric option) Limit a multi-build to just the corresponding line(s) from `$BUILD_MULTI` (see MULTI-BUILDS below).
−n, −−no−run	dry−run — do not actually run the staging functions.
−f, −−force	Remove various safeguards and permit running as root.
−w, −−wipe	Wipe embody log directories (all builds).

General options

−h, −−help	Show option summary.
−q, −−quiet	Suppress trace output (test output is still shown).
−v, −−verbose	Generate verbose output; may be repeated to get increased verbosity.
−−version	Print libembody version number.
−−debug	Generate debugging output.

Available options

EMBODY is normally a shell script and may process its own options. Any options not consumed will be interpreted by *libembody*. Without requiring the use of **−−**, a few alphabet slots are available: **−g**, **−j**, **−k**, **−o**, **−y**. See <http://www.faqs.org/docs/artu/ch10s05.html> for customary meanings.

MULTI-BUILDS

A `BUILD_MULTI` variable specified in *EMBODY* results in several closely related builds. The format is multi-line (requiring enclosing single or double quotes), as follows:

```
# comment
buildtag1   var1=value var2=value ...
buildtag2   var1=value var2=value ...
...
```

Each line defines a value for `BUILD` and several associated variables. *EMBODY* will be called recursively once for each line. During each call `BUILD` will be set to its respective *buildtag* and all associated variables will have their respective values. Empty lines and '#'−style comments in `BUILD_MULTI` are ignored. Setting an explicit value for `BUILD` will *preempt* a multi−build.

VARIABLES

In addition to any variables defined in *./EMBODY*, the following variables are available to staging functions:

`BUILD` (during multi−builds)
: Will be set to each *buildtag* in turn.

`package_build = $VERSION−$BUILD`
: Unique indentifier of the current build; automatically added to the modulefile as Tcl variable `version`.

`package_name = $NAME−$package_build`
: Fully qualified package name.

`prefix = $PACKAGE_ROOT/$package_name`
: Installation destination directory; automatically added to the modulefile as Tcl variable `prefix`.

`embody_logdir = embody−$package_build`
: Workdir for current build logs.

`embody_testdir = test−embody−$package_build`
: Name of a build-specific test directory.

  This is intended to keep a native `test` directory pristine across subsequent builds, should the `make distclean` step be ignorant of it. The directory will be created cleanly for each build; it is up to the user to populate this directory in *stage_test*. After *stage_test*, the directory will be moved to `$embody_logdir/test`. The directory is created initially in the toplevel source directory because some test procedures use relative paths in constructs like `−I../include`.

`module_name = $NAME/$package_build`
: Full module name with version, refers to a file under `$MODULE_ROOT`.

`module_dir = $MODULE_ROOT/$NAME`
: Path to modulefile without version.

`force`, `verbose`, `quiet`
: These variables are non-empty when the corresponding options were specified. Useful for conditionals in user-defined staging functions.

FILES

`$EMBODY_HOME/bin/libembody`
: The **Embody** library.

`<package_name>/EMBODY`
: User-generated **Embody** script.

`$EMBODY_HOME/share/doc`
: Documentation and example files.

BUGS

Options must be given individually (cannot be clustered). This shouldn't hurt too much unless you're running *EMBODY* over and over.

Dry-run mode does not show actions inside staging functions.

SEE ALSO

module (1), *modulefile* (5), *bash* (1), *rpm* (8), *rpmbuild* (8)

<http://trac.anl.gov/embody/>

AUTHOR

Michael Sternberg, Center for Nanoscale Materials at Argonne National Laboratory.

COPYRIGHT

OPEN SOURCE LICENSE

DISCLAIMER